I2C 发生在 Kernel 中的故事

中科龙梦科技有限公司
huhb huhb@lemote.com

October 12, 2012

目 录

§1 背景

故事背景来源于一个丑陋的 Fixup。故事详情是项目中使用到一个温度传感器 TMP75，这个东西使用的如此广泛以至于内核中早就有他的支持了，当然名字不叫 tmp75.c，而是 lm75.c。直接选为 module，然后编译，加载，发现不能正常注册上去。

将 I2C 的 Debug 选项打开，看到设备与控制器的交互过程，在 lm75_detect() 中读取 TMP75 的 4 号寄存器，控制器返回" Error: no response! " 的错误。打开手册，没有看到有这个寄存器，于是将不存在的这些寄存器访问都注销掉。Driver 正常注册，然后通过/sys/class/hwmon/hwmon0/能访问到温度值，读取的值确认也是正确的。

故事到这里才刚刚开始，调试时直接通过将其注销掉的，现在回去看看这地方的为什么会有这么“二”的访问呢?
代码中这个地方有注释，位置在 drivers/hwmon/lm75.c 中的 lm75_detect()，大体意思就是 LM75 芯片没有 ID 寄存器，不能通过 ID 确认在这个地址上的是不是这款芯片。于是想了两个蛋疼的测试来间接确认这款芯片:

- 第一个测试为读取芯片的内部寄存器 4,5,6,7 会返回上一次有效读取的值，
- 第二个测试就是内部寄存器号的 bit7-bit3 变化不会影响到可用寄存器的访问。(补注：新版本的内核已经有说明除了 LM75 和 LM75A，其他兼容的芯片不支持这种方式的探测)

看了 TMP75 的手册，从来就没有这样的描述。只能确定这个是一个 hack 的处理办法，而悲摧的事情是 TMP75 配合 SB710 的 SMBus 控制器不是这样的结果。只能简单的将这段代码使用 `#if 0 . . . #endif` 的方式注销掉。提交 patch 的时候，看到这样的修改真的蛮恶心的。想到在 ARM 平台的时候 I2C 设备都是通过名字进行匹配，看看是否能够修改成这种方式进行处理。于是将 I2C 的子系统看了一遍，确定应该能够通过这种方式处理的!

在 I2C 的整个子系统中，定义了四种软件结构:

- 一种是描述 I2C Master 的，称为 Adapter
- 一种是描述关于 I2C Slave 设备，称为 i2c_board_info
- 一种是控制设备驱动，称为 I2C Driver
- 一种是将 I2C Master 以及 I2C Slave 设备信息整合在一起用于操作的结构，称为 I2C Client

Adapter
　　实现 master 设备的操作，即完成 Driver 的读取，写入 slave 设备的请求。
i2c_board_info
　　描述了 i2c slave 设备使用的地址，设备名称，如果有中断，还包括了中断号等等

Driver
: 完成对设备功能的实现，如 TMP75，Driver 实现对温度的读取以及将结果通过 sysfs 导出到用户空间

I2C Client
: 主要是将设备以及该设备对应的 Adapter 连接在一起，供 Driver 使用的结构;

本来将 I2C 匹配看完就差不多能够完成功能，但是既然题目是发生在 Kernel 中的故事，那么还得有头有尾才行，不然就成了挖坑工了。

如果你看过二十本武侠小说，大致能够理出的一个故事套路就是：主角在前十几或者二十几年中苦练绝技，功成后开始游历江湖，然后碰到很多很多人，做了很多很多行侠仗义的事情，展开了一场波澜壮阔的人生画卷，最后几乎都是左拥右抱钱袋鼓的样子结束。套用到我们这里来就是：

- 先期我们得构建好几个数据结构 I2C Driver，I2C Client，相关的 I2C Adapter，这就是打基础。
- 将这些结构通过相关的 register 方法，注册到设备模型中去，这就是出山。
- 然后想办法让 I2C Device 与 I2C Adapter 相遇，构成一个 I2C Client;
- 然后还得让 I2C Client 与 I2C Driver 相遇，然后能够在 I2C Driver 中使用 I2C Client 处理设备读取相关的东西，这就是故事重要部分;
- 最后确定这个 Driver 只是工作在 Kernel Mode 还是需要用户空间程序去控制，收尾。
- 如果你要展示英雄迟暮的悲凉，你还得关注 unregister 以及 remove，module_exit() 等等东西。

§2 构造数据结构

§2.1 struct i2c_adapter

下面是该结构的定义，在 include/liunux/i2c.h

```
struct i2c_adapter {
    struct module *owner;
    unsigned int class;
    const struct i2c_algorithm *algo;
    void *algo_data;
    struct rt_mutex bus_lock;
    int timeout;                    /* in jiffies */
    int retries;
```

```
    struct device dev;              /* the adapter device */
    int nr;
    char name[48];
    struct completion dev_released;
    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;
};
```

owner
可以直接赋为 THIS_MODULE，这个域能够避免模块在操作的时候被卸载掉。

class
如果 Adapter 需要支持自动扫描的 I2C Driver，这个就需要进行设置。I2C Driver 只能使用同种 CLASS 的 Adapter 进行自动探测。可选的值有：

值	含义
I2C_CLASS_HWMON	lm_sensors
I2C_CLASS_DDC	DDC bus on graphics adapters
I2C_CLASS_SPD	Memory modules

algo
指向 struct i2c_algorithm 结构体指针，实现了具体的数据传输。struct i2c_algorithm 结构如下

```
struct i2c_algorithm {
    int (*master_xfer)(struct i2c_adapter *adap,
                struct i2c_msg *msgs,int num);
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
                unsigned short flags, char read_write,
                u8 command, int size, union i2c_smbus_data *data);

    /* To determine what the adapter supports */
    u32 (*functionality) (struct i2c_adapter *);
};
```

一般实现 smbus_xfer 即可，如果有一些特定的 I2C 操作可能就需要实现 master_xfer 的操作。functionality 也是一个比较重要的函数，他返回主设备能够完成哪些 I2C 操作以及支持的特性。在 I2C Driver 中，我们需要判断 Adapter 支持哪种传输方式，然后能够挑选一个比较优化的方式进行传输。部分值

取值	含义
I2C_FUNC_SMBUS_BYTE_DATA	字节读写
I2C_FUNC_SMBUS_WORD_DATA	双字读写
I2C_FUNC_SMBUS_BLOCK_DATA	块读写

name
定义了 adapter 的名字。

nr
用于创建 bus 号的，即在/sys/bus/i2c/devices/i2c-xx(xx 代表数值从 0 开始递增) 看到的数值。

例子:
drivers/i2c/busses/i2c-piix4.c piix4_adapter 定义

§2.2 struct i2c_board_info

结构体如下 include/linux/i2c.h

```
struct i2c_board_info {
    char            type[I2C_NAME_SIZE];
    unsigned short  flags;
    unsigned short  addr;
    void            *platform_data;
    struct dev_archdata     *archdata;
    struct device_node *of_node;
    int             irq;
};
```

type
: slave 设备名称，会被赋值给 client 的 name 字段，也用于匹配 I2C Driver 中 id_table 必须的

flags
: 标志，会拷贝到 i2c_client.flags 中，这个可以标志一些设备特性比如是否需要 10 地址支持等可选

addr
: slave 设备的 I2C Address 必须的

platfrom_data
: 会将其拷贝到 i2c_client.dev.platfrom_data 可选

archdata
: 会将其拷贝到 i2c_client.dev.archdata 可选

irq
: 如果有中断，可填上中断号可选

例子:
arch/mips/alchemy/board-gpr.c gpr_i2c_info[]

§2.3 struct i2c_client

看看 struct i2c_client 的数据结构 include/linux/i2c.h

```
struct i2c_client {
    unsigned short flags;
    unsigned short addr;
    char name[I2C_NAME_SIZE];
```

```
    struct i2c_adapter *adapter;
    struct i2c_driver *driver;
    struct device dev;
    int irq;
    struct list_head detected;
};
```

flags
: 与 i2c_board_info 中的 flags 相同

addr
: 与 i2c_board_info 中 addr 相同

name
: 与 i2c_board_info 中 type 相同，slave 设备的名字

adapter
: 对应的 I2C Master Driver

driver
: 对应的 I2C Driver

例子：
参考 i2c_new_devices() 的实现，一般来说我们不会直接去填充一个 i2c_client 的结构体

§2.4 struct i2c_driver

再看看 struct i2c_driver 的实现

```
struct i2c_driver {
    unsigned int class;
    int (*attach_adapter)(struct i2c_adapter *) __deprecated;
    int (*detach_adapter)(struct i2c_adapter *) __deprecated;
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);
    void (*shutdown)(struct i2c_client *);
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *);
    void (*alert)(struct i2c_client *, unsigned int data);
    int (*command)(struct i2c_client *client, unsigned int cmd,
                   void *arg);
    struct device_driver driver;
    const struct i2c_device_id *id_table;
    int (*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list;
    struct list_head clients;
};
```

class

与 Adapter 中 class 含义一样，这里代表 Driver 属于哪个类别的。如果 Driver 要求自动探测的话，则需要对应的 Adapter class 含有这个类别。

attach_adapter
: 用于寻找对应的 adapter，然后生成 i2c_client。这个 API 的局限性就是需要预先知道 slave 设备将使用哪个 Adapter。

detach_adapter
: 用于解除对应的引用，可以看到这两个 API 已经标志了__deprecated，所以我们写的时候需要避免使用这两个 API

probe

remove
: 标准的设备驱动模型，如果有匹配的 device，则会调用 probe 函数，注销的时候会调用 remove

driver
: 这个域中 name 代表驱动名，如果与模块名一致，那么系统能够通过 Dbus 动态加载它

id_table
: 支持的 slave 设备名称列表，用于标准驱动设备模型中设备匹配

detect
: 这个函数用于自动匹配。

address_list
: slave 设备可能用到的地址，用于自动匹配的

例子： drivers/hwmon/lm75.c 中 lm75_driver 的定义

§3 匹配的种种情况

接下来看看设备与驱动的绑定情况。I2C 子系统中设备与驱动的匹配大体有两种情况。

§3.1 通过 Linux 设备驱动模型进行匹配

通过注册的设备名称与 Driver 中声明支持的设备名称 (id_tables 中定义的) 是否匹配来完成的。这适合预先已经知道系统中有什么设备的情况。通过两种方法能够实现，其主要的思路就是想法构建一个合适的 i2c_client 结构体
一种编程模型:

```
Example (from omap2 h4):

static struct i2c_board_info __initdata h4_i2c_board_info[] = {
    {
        I2C_BOARD_INFO("isp1301_omap", 0x2d),
        .irq            = OMAP_GPIO_IRQ(125),
    },
    {       /* EEPROM on mainboard */
```

```
        I2C_BOARD_INFO("24c01", 0x52),
        .platform_data  = &m24c01,
    },
    {       /* EEPROM on cpu card */
        I2C_BOARD_INFO("24c01", 0x57),
        .platform_data  = &m24c01,
    },
};

static void __init omap_h4_init(void)
{
    (...)
    i2c_register_board_info(1, h4_i2c_board_info,
            ARRAY_SIZE(h4_i2c_board_info));
    (...)
}
```

通过 i2c_board_info 声明了三个 I2C slave 设备，然后通过 i2c_register_board_info() 会生成三个对应的 i2c_clinet，需要注意到 i2c_register_board_info() 中第一个参数是指的 bus num，这就要求对应的 Adapter 需要固定注册到这个位置。这在嵌入式系统中是比较常见。这种情况适合即知道 I2C Slave 设备又知道 I2C Master 设备情况

第二种编程模型为:

```
Example (from the sfe4001 network driver):

static struct i2c_board_info sfe4001_hwmon_info = {
    I2C_BOARD_INFO("max6647", 0x4e),
};

int sfe4001_init(struct efx_nic *efx)
{
    (...)
    efx->board_info.hwmon_client =
        i2c_new_device(&efx->i2c_adap, &sfe4001_hwmon_info);

    (...)
}
```

这种情况适合我们不能提前知道 bus num 的情况，但是知道有什么设备。还有一种改进型的编程模型

```
Example (from the nxp OHCI driver):

static const unsigned short normal_i2c[] = {0x2c, 0x2d, I2C_CLIENT_END};

static int __devinit usb_hcd_nxp_probe(struct platform_device *pdev)
{
    (...)
    struct i2c_adapter *i2c_adap;
```

```
    struct i2c_board_info i2c_info;

    (...)
    i2c_adap = i2c_get_adapter(2);
    memset(&i2c_info, 0, sizeof(struct i2c_board_info));
    strlcpy(i2c_info.type, "isp1301_nxp", I2C_NAME_SIZE);
    isp1301_i2c_client = i2c_new_probed_device(i2c_adap, &i2c_info,
                             normal_i2c, NULL);
    i2c_put_adapter(i2c_adap);
    (...)
}
```

这种情况适合不知道设备地址在哪个上面的情况，有可能是 0x2c，也有可能是 0x2d 的。i2c_new_probed_device() 只会创建一个有效的 client 或者返回 NULL。这跟后面第 2 种方式的 detect 有区别。

§3.2 通过对设备驱动模型扩展的方式进行匹配

这种方法主要是解决不知道系统中是否有这样的设备而设计的，比如对 PC 上 SMBus Monitor 设备支持。实现方法就是在 I2C Driver 中实现 Address_list 以及 Detect 方法。在调用 i2c_add_driver() 函数中会创建一个临时的 I2C Client，然后使用 Adress_list 定义的地址以 SMBus 字节读方式挨着访问一遍，如果正常回应则是有设备，然后再通过 Detect 方法，进一步判断是否为支持的设备，如果返回成功，则就是有这样的设备了。这个时候开始构造一个 I2C Client，作为 Driver 中 I2C 操作的 Client。在这个模型中如果有多个地址有效，则会创建多个 i2c_client 的。

```
Example :
drivers/hwmon/lm75.c lm75_detect()
```

这种模型中，在 detect 方法中 i2c_client 结构体是一个临时的，而 probe 中的传进来的 i2c_client 就能够用于整个通信了。

这不仅仅是一场约会，而是一生的相守。

§4 工作方式，Kernel Mode 还是需要用户空间去控制

如果是 Kernel Mode 话仅仅调用 smbus_read_byte_data() 就可以了。如果要提供用户态的控制就得挑选方法，或者通过 sysfs，procfs，或者 ioctl，read，write 甚至是 netlink 方式。

§5 后面的一切

其实看到这里，你会发现全篇有点扯淡的感觉。为了不至于太浪费生命，后面的 unregister 以及 remove 就省略了。看看例子就能解决了。

§6 故事的结局

因为知道系统中确实有这样的设备，故通过第一种方式能够实现匹配，而不用去修改 lm75.c 中代码了。最后修正的样子如下：

```
static int __devinit tmp75_probe(struct platform_device *dev)
{
    struct i2c_adapter *adapter = NULL;
    struct i2c_board_info info;
    int i = 0, found = 0;

    memset(&info, 0, sizeof(struct i2c_board_info));

    adapter = i2c_get_adapter(i++);
    while (adapter) {
        if (strncmp(adapter->name, "SMBus PIIX4", 11) == 0) {
            found = 1;
            break;
        }

        adapter = i2c_get_adapter(i++);
    }

    if (!found)
        goto fail;

    info.addr = TMP75_SMB_ADDR;
    info.platform_data = "TMP75 Temprature Sensor";
    /* name should match drivers/hwmon/lm75.c id_table */
    strncpy(info.type, "tmp75", I2C_NAME_SIZE);

    tmp75_client = i2c_new_device(adapter, &info);
    if (tmp75_client == NULL) {
        printk(KERN_ERR "failed to attach tmp75 sensor\n");
        goto fail;
    }

    printk(KERN_INFO "Success to attach TMP75 sensor\n");

    return 0;
fail:
    printk(KERN_ERR "Fail to fount smbus controller attach TMP75
```

```
            sensor\n");

    return 0;
}
```

§7 最好的文档

这篇文章主要是个人学习的一个总结。其实关于 I2C 的文档，内核源码中 Documentation/i2c/下面就有，写的也非常详细了，如果本文与他们有冲突的话，请以内核文档为准，并能够来信提醒我。